To appear in the Proceedings of the Fourth International Symposium on Requirements Engineering (RE'99), Limerick, Ireland, June 7-11, 1999

Formal Modeling of Space Shuttle Software Change Requests using SCR

Virginie Wiels
ONERA-CERT
BP 4025, 2 avenue Edouard Belin
F31055 Toulouse, France
Virginie.Wiels@cert.fr

Steve Easterbrook
Institute for Software Research
1000 Technology Drive,
Fairmont, WV 26554, USA
easterbr@csee.wvu.edu

Abstract

This paper describes a feasibility study into the use of a formal requirements modeling method (SCR) to assist with Independent Verification and Validation of change requests for Space Shuttle flight software. The goal of the study was to determine whether a formal modeling technique could automate some of the manual analysis tasks performed on change requests, including consistency checking. To analyze the change request, the key part of the original functionality was modeled in SCR. The model was then updated to reflect the proposed changes. Tool support was used to perform consistency checking and to validate the model against domain properties. The study showed that as an analysis tool, formal modeling offers some advantages over inspection-based approaches. However, the problem of analyzing change requests is sufficiently different from other requirements modeling tasks that some specialist tools will be needed. The paper ends with a discussion of the demands of these needs.

1. Introduction

This paper reports on our experiences applying the SCR ("Software Cost Reduction") method [1] to model flight software change requests for the space shuttle, as part of an investigation of automated tools for software verification and validation. This study explored the applicability of SCR for determining correctness, consistency, and completeness of shuttle change requests. The work was conducted at the request of the Shuttle Independent Verification and Validation (IV&V) team. The IV&V team was interested in SCR for two reasons:

- The ability to automate some of the tedious manual consistency checking that is currently necessary when analyzing change requests.
- The ability to animate the requirements in a simulation, to validate the specified behavior.

One of the key questions addressed by this study is whether formal methods can be applied to reason about requirements changes to legacy systems for which there is no existing formal specification. Even if good quality documentation exists for such systems, the problem of tracing the impacts of proposed changes is immense. For such systems, the ability to automate some aspects of verification and validation of change requests may represent a large cost saving. We regard focussed application of formal methods as a step towards providing that automation.

Formal modeling of change requests for a large system is interesting for a number of reasons. Firstly, very little attention has been paid in the formal methods community to the management of change during systems development. Hence, most case studies of formal methods have concentrated on the use of a formal specification as a baseline from which designs and implementations can be verified. Consideration of how changing requirements are handled in this process have been limited to dealing with *refinement* (i.e. correctness-preserving elaborations), such that if the implementation is derived from a specification, then an updated implementation can be derived in a similar way from a refined specification. In large systems development, many requirements changes occur that are not correctness preserving. If formal methods are to be successfully applied in such a context, they must be able to handle evolving specifications, without requiring extensive effort to modify and re-validate the formal models. In principle, formal methods open up new possibilities for reasoning about proposed requirements changes, thus helping to validate the change requests, and hence facilitate the decision about whether to accept or reject each proposed change.

Space Shuttle change requests offer an excellent testbed to explore these issues. The shuttle flight software is an extremely well documented system, where a complete set of revised specifications is available for each version of the software, and the lifecycle of each change request is fully documented. The process for proposing

and evaluating change requests is relatively mature, so that the relative benefits of any new techniques can be measured with some accuracy. Finally, the current specifications are very detailed, but entirely informal.

1.1 Shuttle Change Requests

As an operational vehicle, the Space Shuttle regularly needs updates to its flight software to support new capabilities (such as docking with the space station), replace obsolete technology (such as the move to GPS for navigation), or to correct anomalies. Software updates are known as Operational Increments (OIs), and are typically completed approximately every twelve to eighteen months. An OI will implement any number of change requests (CRs). Each change request goes through a rigorous analysis and review process before it can be approved for inclusion in an OI. A change request typically consists of a selection of pages from current Functional Subsystem Software Requirements (FSSR) specifications, with handwritten annotations showing new and changed requirements. Change requests vary in length from a few pages to several hundred pages.

Each change request is reviewed by a number of requirements analysts, along with members of the IV&V team, culminating in a formal requirements inspection. Following the inspection, the change request may be rejected, revised for re-inspection, or forwarded to the review board for inclusion in the current OI. The Shuttle Avionics Software Control Board (SASCB) makes the final decision whether to include each CR in the OI. Their decision takes into account various factors, including total size of the changes, relative priorities of the change requests, and interaction between change requests.

This case study concentrated on change request #90724, the East Coast Abort Landing (ECAL) automation. The change request consisted of 104 pages, affecting seven existing specifications. The change request covers changes needed to automate the entry guidance for an emergency landing at sites on the East Coast or Bermuda. Several types of abort are possible following a loss of thrust during launch, including return to launch site (RTLS), transatlantic abort (TAL) and abort to orbit. ECAL is an intermediate solution between RTLS and TAL, but is not currently automated. The rationale for automating ECAL is that it will reduce costs of crew training, and increase the probability of successful abort landing. The additional functionality includes the management of the shuttle's energy during descent and the guidance needed to align it with the selected runway.

1.2 Modeling approach

SCR was chosen for a number of reasons:

- SCR was designed for reactive control systems, indicating a good match with shuttle guidance and navigation software.
- Tool support was available, with extensive automated consistency checking and a simulator linked to a model checker to validate dynamic properties of the model.
- The basic constructs of SCR match the shuttle documentation very well. The use of tables to represent control functions and the ability to include real-valued input variables directly in the model help to reduce the conceptual distance between the existing documentation and the formal model.

Initial analysis indicated that it would not be feasible to model the new functionality of the change request in isolation from the existing requirements. Accordingly, our approach was to model the existing requirements first, checking this model for consistency, and then updating it to reflect the changes listed in the CR.

In order to focus the modeling effort, we concentrated on only one of the seven FSSRs affected by the change request, namely STS-83-0001-27 "GN&C Part A Entry Through Landing Guidance". The changes all referred to the "Return-to-launch site (RTLS) Terminal Area Energy Management (TAEM) Guidance". This represented approximately two-thirds of the page count of the change request. This portion was selected because it contained the core of the new functionality, whilst allowing us to restrict the scope of our modeling.

2. SCR

SCR (Software Cost Reduction) is a formal method for modeling and validating system requirements as a deterministic state machine, represented using a set of tabular notations. A toolset has been developed by the Naval Research Laboratory (NRL) to support the method. In this section, we give a brief description of the SCR method and tool. Detailed descriptions of the syntax and semantics of SCR can be found in [1].

2.1 SCR method

SCR models a system as a black box that computes output data from input data. Three kinds of variables are distinguished: *monitored variables* represent the inputs of the system (environmental quantities that influence system behavior); *controlled variables* represent the outputs of the system (environmental quantities that the system controls); and *terms* are intermediate, internal variables used in the computation. The functionality of the system is defined by expressing the values of the controlled variables as a function of the monitored variables, terms, and current system state. To facilitate this definition, SCR introduces two other modeling primitives: *mode classes* and *events*. Mode classes

represent abstract states in the system behavior. There can be several different mode classes for the same system: each mode class has a name, possible values ('modes') and an initial mode. Events are changes in value of variables. Each change in a monitored variable is treated as a separate event, and a simplifying assumption is made that only one input event occurs at once (i.e. that it is always possible to compute the new state arising from an event before processing the next event).

An SCR specification consists of a set of dictionaries of vocabulary elements, and a set of tables defining the system behavior. Each type of vocabulary element is listed in its own dictionary, one for each of: variables, types, constants and mode classes. Each mode class, term, and controlled variable is then defined in its own table. The value of a mode class is expressed as a deterministic finite state machine in a *mode class table.* The table gives the next mode for each mode change, according to the current mode and the event that triggers the mode change. The value of each output variable (and of each term) of the system is described either as a *condition table* or an *event table,* such that each output variable has exactly one table defining it:

A condition table defines the values of a variable for each mode of the associated mode class. Rows represent the modes, while columns represent the values the variable can take. Each cell gives the condition under which the variable takes the given value in the given mode. A condition table is well-formed if it has the following two properties:

1. Completeness: the value of the variable must be defined in every possible system state.
2. Disjointness: in any situation the table gives a unique value for the variable.

An event table is slightly different: it describes the cases where the value of the variable *changes*. If the system is in a given mode and a certain event occurs, the event table gives the value that the variable will take in the next state. Event tables must have disjointness, but not completeness.

The choice of whether to use an event or a condition table for each controlled variable is a modeling decision, as is the choice of which mode class to associate with each table. We used event tables almost exclusively, as these matched the operational nature of our source documents. We discuss this issue further in section 6.

Finally, at the semantic level, an SCR specification represents a single deterministic state machine, where the system state is a vector of the values of all environment variables (monitored and controlled), along with a current value for each mode class.

The NRL toolset includes a specification editor, a simulator for symbolically executing the specification and formal analysis tools for verifying selected properties. The tool includes a consistency checker, for specification syntax and type checking, for checking the completeness and disjointness of tables, detecting dependency cycles and checking that every mode in a mode class is reachable. The tool also supports assertions, which are conditions on individual states or on a pair of adjacent states, and are proved using a model checker or theorem prover [2]. The simulator allows the user to 'run' a specification, by entering a sequence of input events, and observing the resulting outputs.

3. RTLS TAEM Guidance

We modeled the ECAL change request in two steps. First, we specified the existing requirements for Return-To-Launch-Site (RTLS) Terminal Area Energy management (TAEM) Guidance in SCR. Then, we updated this SCR model to add the change request concerning East Coast Abort Landing (ECAL).

3.1 RTLS TAEM Guidance

The RTLS TAEM Guidance specifies the software functions required to guide the orbiter during a contingency abort. Descent is unpowered: the orbiter acts as a glider. Acceleration is controlled by varying the angle of attack of the orbiter. The specification defines seven guidance phases (numbered in the sequence: 6, 5, 4, 0, 1, 2, 3). The RTLS phases, 6 (alpha recovery), 5 (normal acceleration (NZ) hold), and 4 (alpha transition phase), are the concerned with bringing the orbiter under control with respect to angle of attack, acceleration and attitude. The TAEM phases, 0 (S-turn phase), 1 (acquisition phase), 2 (heading alignment phase), and 3 (prefinal phase), are concerned with aligning the orbiter with the selected runway, and controlling the approach. Two extreme conditions have to be handled. If considerable excess energy exists, an S-turn maneuver is executed to dissipate this energy. If the vehicle has extremely low energy, the guidance software sets a flag, to request that the crew switch the heading alignment cone (HAC) to the minimum entry point (MEP) HAC location. Crew action is required to achieve relocation of the HAC. The RTLS TAEM guidance software receives input from the inertial navigation and air data subsystems. A ground track predictor routine (GTP) estimates the horizontal distance to the runway threshold. This range prediction is used to calculate energy, altitude, altitude rate and dynamic pressure values.

The RTLS TAEM specification is divided into fourteen functions. An executive routine (called GREXEC) is executed at each time cycle. This routine calls thirteen other functions depending on the value of the variable IPHASE, representing the current guidance phase:

- Guidance initialization (GRINIT)
- RTLS Phase transition, and low energy alert (GRTRN)

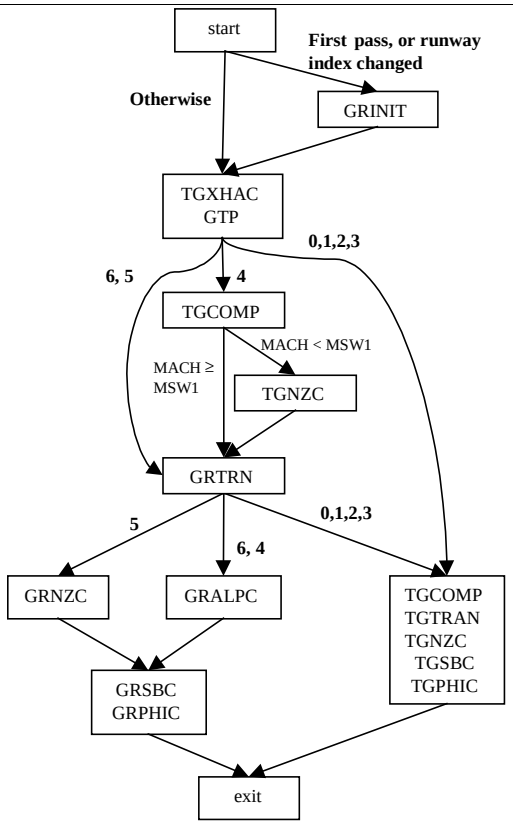

Figure 1: A call graph for the GREXEC function. Labels on the arcs are conditional branches; numbers refer to the current guidance phase.

- RTLS Angle of attack command (GRALPC)
- RTLS Normal acceleration command (GRNZC)
- RTLS Speed brake command (GRSBC)
- RTLS Roll command (GRPHIC)
- Heading alignment cone location function (TGXHAC)
- Ground Track Predictor (GTP)
- TAEM References, dynamic pressure, and spiral adjust (TGCOMP)
- TAEM Phase transition and low energy alert (TGTRAN)
- TAEM Normal acceleration command (TGNZC)
- TAEM Speed brake command (TGSBC)
- TAEM Roll command (TGPHIC)

The sequencing of these functions is described in natural language in the documentation; figure 1 presents this sequencing graphically.

Each of these functions is described in a similar way:

1. general requirements in natural language;
2. detailed requirements in pseudo-code;
3. tables of input and output variables;
4. tables of constants, I-loads, and K-loads[1];
5. other requirements, e.g. initialization.

We will take one of these functions (GRALPC) as an example. The GRALPC function computes the angle-of-attack for the alpha recovery phase (IPHASE=6) and the alpha transition phase (IPHASE=4). Detailed requirements are given in figure 2.

If IPHASE=6, the constant alpha recovery angle-of-attack command, ALPCMD, as well as the altitude rate dependent incremental NZ command, DGRNZ, for the load relief phase (IPHASE = 5) are computed as shown in Equation Set 1.

1.1 If CONT=OFF then ALPCMD=ALPREC

*1.2 If CONT=ON then ALPCMD=MIDVAL(ALPRECS * MACH + ALPRECI, ALPRECU, ALPRECL)*

1.3 If HDOT<HDMAX, then HDMAX=HDOT Otherwise (HDOT>=HDMAX) execute Equation Set 1.3 for intact aborts (CONT=OFF) or Equation Set 1.4 for contingency aborts (CONT=ON)

1.3.1 DGRNZ= MIDVAL((HDNOM – HDMAX) DHDNZ, DHDLL, DHDUL)*

1.3.2 DGRNZT= GRNZC1 + DGRNZ + 1.0

*1.4.1 DGRNZT= MIDVAL(DNZB – HDMAX * DHDNZ, DNZMIN, DNZMAX)*

*1.4.2 SMNZ1= ZDT1*DGRNZT*

1.4.3 DGRNZ= DGRNZT – GRNZC1 – 1.0

1.4.4 NZSW= GRNZC1 – SMNZ1 – SMNZ2 + 1.0

Otherwise (IPHASE /= 6), the angle-of-attack command for the alpha transition phase is computed. Equation Set 2 tests for the initial pass through the logic and initializes the command.

2.1 If IGRA=0, then ALPCMD=ALPHA and IGRA=1

Next, a test on IGRA is made. If IGRA = 1, Equation Set 3 is executed to compute the smoothed angle-of-attack command and the function is exited.

3.1 DGRALP=MIDVAL(GRALPR–ALPHA, GRALL, GRALU)

3.2 ALPCMD=ALPCMD+DGRALP

3.3 If (DGRALP < 0.0 and ALPCMD <= GRALPR) or (DGRALP > 0.0 and ALPCMD > GRALPR), then ALPCMD = GRALPR and IGRA = 2

Otherwise (IGRA /= 1), Equation Set 4 is used to set the angle-of-attack command equal to the reference angle of attack

4.1 ALPCMD = GRALPR

The function is exited.

Figure 2: Detailed requirements for GRALPC.

3.2 ECAL Change Request

The goal of the change request is to automate East Coast and Bermuda abort landings. This will decrease the loading on the vehicle, increase survivability of the aborts and increase capability to reach a landing site. The change request consists of a list of changes to the previous functions, mainly additions of variables and modifications of requirements.

For example, the changes for the GRALPC function are given in figure 3. Five new constants (DPSAC1, DNZMX1, DNZ1, TLFMX1 and GRALU2), and the three new

[1] *constants are physical constants; I-loads are mission dependent parameters, and can be considered to be constant for each flight; K-loads are constant across a number of flights*

variables (DPSAC, LOAD_TOTAL and ECAL) are added.

4. Specification with SCR

Our initial problem was whether to write one SCR specification for each function or one SCR specification for the whole system. In the former case, we would not be able to verify any global property or simulate the behavior of the whole system, as there is currently no way of composing individual SCR specifications. Our main concern was with the integrity of the whole system, so we chose to write a single SCR specification for the system.

4.1 Modeling the Existing Requirements

We created the SCR specification as follows:

1. Type dictionary. We created a new type for each different unit of measurement. We could just have typed all the variables as Float, but we would have lost the information about units. For example:

Name	Base Type	Units	Legal Values
T_sec	Float	sec	[-10000.0,10000.0]

   We also created an enumerated type *Flag* for the flag variables. There is no unit associated with this type:

Name	Base Type	Units	Legal Values
Flag	Enumerated	ND	ON, OFF

2. Variable dictionary. For each variable, we considered whether it was an input or output for the entire system. Those that are used only for communication between functions were modeled as terms. We used the comments field for traceability to the functions. For example:

Name	Class	Type	Initial Value	Comment
ALPCMD	Cont	T_deg	0.0	GRALPC
DGRNZ	Term	T_g	0.0	GRALPC
SMNZ1	Term	T_g	0.0	GRALPC, GRNZC, GRINIT

   Providing initial values for all variables proved hard, as these were not always given in the specifications.
3. Constant dictionary. We modeled the constants, I-loads and K-loads as SCR constants. Providing values for the I-loads proved hard. As I-load values are not defined in the specification, we used the set of values that are made available for testing purposes on the shuttle project web site. For example, we have:

Name	Type	Value	Comment
ALPREC	T_deg	50.0	GRALPC
DHDLL	T_g	-0.1	GRALPC
DNZB	T_g	0.65	GRALPC

4. Mode classes. We identified two SCR mode classes. The first represents the seven guidance phases, expressed in the requirements by the IPHASE variable. The other mode class was somewhat unnatural for SCR: we chose to represent the current function executed by the system as a mode class. This turned out to be very useful, as we could not otherwise model the functional decomposition in SCR. The table for the phase mode class was built from all the functions that describe phase transitions. The table for the function mode class is in fact the description of the GREXEC routine.
5. Condition and Event Tables. We wrote a table for each controlled variable and term of the system.

Renumber 1.4.2, 1.4.3 and 1.4.4 respectively 1.4.4, 1.4.5 and 1.4.6 and add:

1.4.2 If (ECAL=ON and ABS(DPSAC) > DPSAC1 and DGRNZT < DNZMX1) then ITGTNZ = 1

1.4.3 If ITGTNZ=1, then DGRNZT = DGRNZT+DNZ1

Renumber 3.1, 3.2, 3.3 respectively 3.2, 3.3, 3.4 and add the following just before 3.2:

If (ECAL = ON and LOAD_TOTAL < TLFMX1) then Equation 3.1 is executed and Equation 3.2 is skipped. Otherwise, Equation 3.1 is skipped and Equation 3.2 is executed

3.1 DGRALP=MIDVAL(GRALPR–ALPHA, GRALL, GRALU2)

Figure 3: The changes to GRALPC. In the CR this appears as a set of hand annotations to the original specification. We have modified the wording slightly so we can present it separately.

The main modeling effort lay in this last step: creating the tables for controlled variables and terms. The method we used was as follows. Given a variable, we first scan the FSSR to find the functions in which this variable is modified. We then analyze the different requirements in order to decide which kind of SCR table is needed. Finally we build the table. We take two examples to illustrate this method: SMNZ1 and ALPCMD. SMNZ1 is modified in three functions: GRINIT, GRALPC and GRNZC. In GRINIT, SMNZ1 is initialized to SMNZC1. The requirements for GRALPC are given in figure 2, while the relevant section of GRNZC is the following:

Upon entering GRNZC, Equation Set 1 is executed to calculate the time-variant shaping parameters SMNZ1 and SMNZ2, and then these are used to calculate the normal acceleration command NZC.

*1.1 SMNZ1 = SMNZC3*SMNZ1*

1.2 SMNZ2 = SMNZ2 – SMNZC4

1.3 If SMNZ2 < SMNZ2L, then SMNZ2 = SMNZ2L

1.4 NZC = GRNZC1 – SMNZ1 – SMNZ2 + DGRNZ

We can extract from these three functions the requirements concerning SMNZ1. They are as follows:

GRINIT: If INIT_PASS=OFF then SMNZ1= SMNZC1

GRALPC: If IPHASE=6 and HDOT>=HDMAX and CONT=ON, then SMNZ1 = ZDT1*DGRNZT

GRNZC: SMNZ1 = SMNZC3*SMNZ1

We then have to decide whether to write a condition or

Modes	Events		
GRINIT	NEVER	**@T**(Inmode) **when** (INIT_PASS=OFF)	NEVER
GRNZC	**@T**(Inmode)	NEVER	NEVER
GRALPC	NEVER	NEVER	**@T**(Inmode) **when** (IPHASE=6 AND HDOT >= HDMAX AND CONT=ON)
SMNZ1' =	SMNZC3 * SMNZ1	SMNZC1	ZDT1 * DGRNZT'

Figure 4: The Event table for SMNZ1. The notation @T(c) represents the event 'condition c becomes true'. *Inmode* is used as shorthand for entry into the mode.

an event table for SMNZ1. In this case, there is only one possible solution. Indeed, the new value of SMNZ1 in GRNZC is computed using the old value of SMNZ1; we thus have to use an event table. Also, SMNZ1 is given a new value only under certain conditions, so we would not be able to write a complete condition table. The table for SMNZ1 is shown in figure 4. Note that in the final row, the result uses DGRNZT' because DGRNZT is computed just before SMNZ1 and this is the new value that we need in order to compute SMNZ1.

ALPCMD is computed only in GRALPC. The requirements for ALPCMD can be summarized as:

If IPHASE = 6 then
If CONT = OFF then ALPCMD = ALPREC
If CONT = ON then ALPCMD = MIDVAL
*(ALPRECS*MACH + ALPRECI, ALPRECU, ALPRECL)*
If IPHASE /= 6 then
If IGRA = 0 then ALPCMD = ALPHA and IGRA = 1
If IGRA = 1 then
ALPCMD = ALPCMD + DGRALP
If (DGRALP < 0.0 and ALPCMD <= GRALPR)
or (DGRALP > 0.0 and ALPCMD > GRALPR
then ALPCMD = GRALPR and IGRA = 2
If IGRA /= 1 then ALPCMD = GRALPR

The problem we have to face here is sequence. The requirements are written in pseudo-code and often use the fact that instructions are executed one after another. For example, here, if IPHASE ≠ 6 and IGRA = 0, we have ALPCMD = ALPHA, then ALPCMD = ALPCMD + DGRALP. After that, a test is made on the value of ALPCMD that has just been computed and ALPCMD may be given yet another value. In this case that ALPCMD is being used to store intermediate calculations.

Expressing this in SCR is not straightforward. We are obliged to flatten the sequential computations expressed in the requirements. For example, for the last test on the values of DGRALP and ALPCMD, we cannot simply use DGRALP and ALPCMD because these refer to the old values of the variables; we should use DGRALP' and ALPCMD'. However, primed variables are not allowed in the event part of the table. We thus have to explicitly write MIDVAL(GRALPR – ALPHA, GRALL, GRALU) for DGRALP' and ALPCMD + MIDVAL(GRALPR – ALPHA, GRALL, GRALU) for ALPCMD'. Finally, these expressions can be simplified by defining them as terms.

The complete table for ALPCMD is given in figure 5. We use the phase mode in this table. We could also have used the function mode but it would have been less suited because ALPCMD is only computed in one function.

4.2 Modeling the Changes

Once the SCR model of the existing requirements was available, modeling the change request was relatively straightforward. The new variables and I-loads were added to the variable and constant dictionary. For each new variable, a table was defined, using the same process as described above. In some cases, the addition or deletion of equations led to changes in the existing tables. Typically, the addition of a new equation requires the addition of a column to the event or condition table representing the variable. In some cases the changes were harder to trace, as they represented intermediate equations in a sequence that had been 'flattened' when it was

Modes	Events				
ph6	**@T**(FN=GRALPC) **when** (CONT=OFF)	**@T**(FN=GRALPC) **when** (CONT=ON)	NEVER	NEVER	NEVER
ph5,ph4,ph3, ph2,ph1,ph0	NEVER	NEVER	**@T**(FN=GRALPC) **when** ((IGRA=0) and not(COND))	**@T**(FN= GRALPC) **when** (((IGRA=1 or IGRA=0) and COND) or IGRA=2)	**@T**(FN= GRALPC) **when** (IGRA=1 and not(COND))
ALPCMD'=	ALPREC	MIDVAL(ALPRECS * MACH + ALPRECI, ALPRECU, ALPRECL)	ALPHA + DGRALP'	GRALPR	ALPCMD + DGRALP'

Figure 5. The event table for ALPCMD. The expression *COND* is defined as (DGRALP_PRIME<0.0 and ALPCMD_PRIME<=GRALPR) or (DGRALP_PRIME>0.0 and ALPCMD_PRIME>GRALPR).

modeled in SCR. In such cases, better traceability between the original specification and the SCR model would be helpful – in some cases we ended up having to reconstruct the process by which the SCR model was derived, in order to reason about how a change should be modeled.

In effect, to model the change request, we modeled the entire functionality as it would be once the change is applied. The anticipated benefits of formal modeling are then available: the model can be automatically checked for consistency, the simulator can be used to observe the behavior under different conditions, and assertion checking can be used to verify key properties. However, this approach did not allow us to model nor reason about the changes in isolation from the original requirements. This has two drawbacks. Firstly, errors detected in the new model have to be traced back to determine if they are errors in the original specification, or errors in the change request. This is also true of our own modeling errors. Secondly, it means that a much larger formal model needs to be constructed than we had hoped, as we cannot just model the changes.

5. Results

We modeled the RTLS TAEM Guidance requirements, and modified the model according to the ECAL change request. We obtained a specification with 30 types in the type dictionary, 2 mode classes, 265 constants, 165 variables, 2 mode class tables, 46 controlled variable tables and 84 term tables. The SCR consistency checker discovered a number of typographic and syntactical errors and unspecified variables that were errors made during our modeling process. As with earlier studies [3], the fact that the consistency checker automatically detects such errors gives us greater confidence that the final model is faithful to the original specification.

A number of defects occurred in both the original specifications and the change request. We found these defects in two ways: ambiguities found during modeling, and inconsistencies found using the SCR toolset. These were as follows:

Ambiguities: An important systematic ambiguity was detected in the FSSR. The detailed requirements are given as a kind of pseudo-code and there are a number of conditional branches. The ambiguity arises when the end of a conditional branch is not clearly indicated. For example, the sequence of execution in GREXEC contains the phrases “If IPHASE ≥ 4 then logic set 1 is executed”; “After logic set 1 is executed, another test on IPHASE is performed ...”, and so on. However, it is not clear whether the latter means the test is performed only if logic set 1 was executed, or whether it should be performed anyway. For the diagram in figure 1 we assumed the latter. This type of ambiguity occurs throughout the requirements. Furthermore, it also occurred in the change request. For example, in figure 3 a condition is introduced to select between equations 3.1 and 3.2. It is not clear then whether equations 3.3 and 3.4 should then be executed in both cases. This particular error was also detected in the inspection process, and was corrected in later revisions of the change request.

Missing initial values: SCR requires all variables to have defined initial values. The original specification only lists initial values for those internal variables (i.e. terms in SCR) for which initialization is needed to ensure correct functioning. It could be argued that a missing initial value is not an error in the specification unless there is a variable that is used before it is assigned a value. SCR did not allow us to distinguish such cases.

Type errors: The specification sometimes used ‘true’ and ‘false’ instead of ‘on’ and ‘off’ for a variable defined as a flag. This is a documentation error in the original specification. Note that in HAL/S, the programming language used for Shuttle avionics, these values are synonyms. Inconsistent use in the specification is considered a question of style, rather than an error.

Modified constants: The shuttle specifications make use of *constants*, *I-loads*, and *K-loads*. Constants tend to be used only for fundamental physical constants. I-loads are mission-specific parameters (e.g. payload weight) or flight parameters for which optimal values have not yet been determined. K-loads are flight parameters that are held fixed over a number of missions. In our SCR model, we modeled all of these as constants. Consistency checking then revealed that some I-load values are modified in the detailed requirements. This is not regarded as an error by the requirements analysts, but happens sufficiently rarely that it is useful to be able to detect occurrences automatically.

Dependency cycles: A dependency cycle was detected between variables PHIC and PHILIMIT. PHIC is computed using PHILIMIT when IPHASE = 0 and PHILIMIT is computed using PHIC value when IPHASE = 3. This is not a real dependency cycle because both variables can be calculated in every case. If IPHASE = 0, PHILIMIT is first given a value, then PHIC is calculated using that value. If IPHASE = 3, it is the other way round. SCR checks for cycles by building for every variable, *v*, a dependency list containing all the variables needed for the computation of *v*. This list is not parameterized by different conditions. The checker consequently detects a cycle. We removed this cycle by defining two additional variables:

- PHICbis has the same value as PHIC but is only computed when IPHASE = 3. It is used by PHILIMIT but does not depend on it.
- PHILIMITbis has the same value as PHILIMIT but is only computed when IPHASE = 0. It is used by PHIC but does not depend on it.

Coverage errors: Coverage errors exist when a condition

table does not define the value of a variable for all possible conditions. We found that coverage errors occurred for two reasons. Firstly, they represented cases where the value of a variable is given for some conditions, with the assumption that it keeps its old value otherwise. This would normally be represented as an event table in SCR, except that we found cases where there was no event (and therefore no state change) associated with the variable. This issue is discussed further below. The second case is when the value is left undefined under some conditions because those conditions are expected never to occur. This case cannot be represented in SCR, which means that we lose the opportunity to check automatically that the undefined state never does occur.

The majority of these findings would not be considered errors by shuttle requirements analysts. In some cases (e.g. modification of I-loads) it is useful to be able to automatically detect them, as they may lead to errors. It is also very useful to be able to automate many of the tedious manual consistency checks applied by IV&V analysts. Moreover, failure to find many inconsistencies was not surprising: this is a mature specification.

6. Discussion

In this section, we discuss our observations on the suitability of SCR for analysis of this type of change request. We first give an overall view of the advantages and disadvantages of applying SCR then detail one particular issue: the need to represent structure.

6.1 Benefits and difficulties of SCR

In this study, we used SCR for an analysis task that is slightly different from that for which SCR was intended. SCR has been used both for initial modeling of requirements, and for reverse engineering requirements models from detailed specifications [1]. However, in either case, the normal approach is to develop a formal specification as a *replacement* for an informal specification: emphasis is placed on establishing the correctness of a formal model, rather than the correctness of an existing informal specification. In our study, we used SCR to analyze (changes to) an informal specification without replacing it. This implies a need for greater flexibility in structuring the formal model, to allow a better mapping between the informal specification and the formal model.

In many ways, SCR is well suited to the task. The organization of the SCR specification followed naturally from the organization of the existing documentation, where tables of input and output variables are already given for each function. Furthermore, the automated consistency checking tool proved to be extremely useful for debugging our model, and for revealing defects in the original specification.

However, there were a number of difficulties that arose from the gulf between the conceptual model on which SCR is based, and that used in the informal specification that we were analyzing:

(1) We found it a difficult abstraction step from the sequential processing narrative to a state machine model. A single state change in the SCR model often represents a long series of calculations in the detailed specification. In some cases we had to introduce artificial states to facilitate the modeling process.
(2) We found that a precise understanding of the detailed semantics of SCR was needed to avoid or remove dependency cycles.
(3) Traceability between the original specification and the SCR model became a problem as the model grew.
(4) The main structure of FSSR (a functional decomposition) could not be expressed in the SCR model, hence some of the consistency checking of the FSSR that we had hoped to do was not possible. We further discuss this issue below.

In addition to these issues, the current toolset can be regarded as a research prototype. It imposes some restrictions on the expressiveness of the SCR model, which we anticipate can be addressed relatively easily. Briefly, these are: (1) the ability to use ‘undefined’ as a value for any variable. This would allow us to accurately model situations where no initialization value is necessary, as our current solution of picking arbitrary initialization values can mask errors in the specification. (2) the ability to use ‘unchanged’ in condition tables, or symmetrically, to add a ‘continuously’ event to event tables, as proposed in [4]. This allows a hybrid between condition and event tables, for variables that are updated in response to events in some places, and continuously in others. (3) The ability to define support functions, such as trigonometric functions, in the tool. The NRL team has added some of these functions to the tool at our request, but a general solution is needed for including functions in an SCR model.

6.2 Structuration issues

The most important disadvantage of SCR in this case study is the lack of structuring primitives. Given the way the requirements are expressed, we would have liked to write one SCR specification for each function and then to compose these specifications to get the system specification. The SCR method does not provide this type of compositionality and we were consequently forced to write one single specification for the whole system. This introduced several problems.

Firstly, we found it hard to maintain traceability between the SCR specification and the original

requirements. Each variable of the system is defined by an SCR table, but this table does not indicate in which function(s) the variable is computed. In practice, we used two methods to record links from the SCR specification to the original requirements: the function mode and the comment part of the variable dictionary. Moreover, the SCR specification obtained for the whole system is quite large, and so is not easy to read and understand.

Secondly, the interconnections between the different functions cannot be checked in the SCR model. These interconnections correspond to dataflows across a functional decomposition. As SCR does not allow these to be expressed, we would need to use a separate modeling tool to check consistency of these.

Finally, structure is important to deal with changes. Structure can be used to isolate changes, and to allow some parts of a specification to be modified more easily. Structure also provides a framework for comparing two versions of a specifications. A specification that is well structured is more robust in the face of change.

In [4], an experience using CoRE (Consortium Requirements Engineering) to specify a flight guidance system is reported. A CoRE specification consists of two parts: a behavioral model similar to SCR and a class model that superimposes an object-oriented organization on the behavioral model by grouping portions of the specification that are related and likely to change. The authors conclude that the class model is very useful to organize a specification and make it robust in the face of change. As yet, the CoRE method has no formal semantics and no associated tool.

A structuration like the one proposed in CoRE is an interesting first step. It addresses our concerns with readability and traceability. However, the structuration in CoRE is only syntactic: some parts of the global specification are grouped into classes, but the monitored and controlled variables are all global with respect to the system. Hence a system specified in CoRE cannot be reused as a component of a bigger system.

Ideally, we would like to be able to define SCR components, to interconnect them, and to compose them to derive a new SCR specification, describing the whole system. In this way encapsulation can be achieved within each component specification, without limiting the compositionality to one or two levels. Verification could be performed on each SCR component and on the interconnections in order to decrease the number of checks that have to be done on the global specification.

7. Conclusions and Future work

Our main conclusion from this study is that although SCR is well adapted to modeling this type of system, and provides a prototype tool to perform consistency checking, it is still immature in some important respects for widespread adoption. There are two areas that we feel need further attention, namely mechanisms for structuring SCR specifications by composing specifications for individual components, and mechanisms for the management of change, so that different versions of a specification can be stored, and the relationships between them analyzed.

To address these issues we are proceeding with several related studies. Firstly, we are exploring how to handle the structuration in SCR. We have examined a framework for composing SCR specifications, and are currently implementing a prototype tool to permit this. As an initial test of this tool, we will develop a model of the requirements described in this case study, broken down into separate models for each function.

Secondly, we are re-visiting our initial decision to use SCR for this type of analysis. It is possible that the problems we encountered might not occur with other modeling methods. Accordingly we plan to model the same set of requirements using techniques such as RSML [5]. RSML is an interesting choice because it includes some of the composition primitives that we would like to see added to SCR. In particular, sequential composition of RSML specifications has been proposed and explored. By applying RSML to the same case study, we hope to obtain a detailed comparison of the two methods, especially with respect to their ability to handle structure and change.

8. Acknowledgements

We would like to thank John Bradbury and Vince Shah of the Shuttle IV&V team for assistance with the modeling, and Bruce Labaw and Connie Heitmeyer for assistance with the SCR tool. This work was partially supported by NASA grant NAG 2-1134.

9. References

[1] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 231-261, 1996.

[2] C. L. Heitmeyer, J. J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications," *IEEE Transactions on Software Engineering*, vol. 24, 1998.

[3] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling," *IEEE Transactions on Software Engineering*, vol. 24, pp. 1-11, 1998.

[4] S. Miller and K. Hoech, "Specifying the mode logic of a flight guidance software in CoRE," *Formal Methods in Software Practice*, Clearwater Beach, Florida, USA, March 4-5, 1998.

[5] M. Heimdahl and N. Leveson, "Completeness and Consistency Analysis of State-Based Requirements," *IEEE Transactions on Software Engineering*, vol. 22, pp. 363-377, 1996.